What is SIFT(Scale Invariant Feature Transform) Algorithm?

Aishwarya Singh

Last Updated : 07 Apr, 2025

103

Ever wonder how you instantly recognize a friend's face, even from a blurry photo or at a different angle? It's because your brain excels at identifying key features and making connections. The Scale-Invariant Feature Transform (SIFT) algorithm, a powerful tool in computer vision, mimics this ability.

Unlike humans, machines struggle to recognize objects in images with variations in scale or perspective. SIFT bridges this gap. By extracting distinctive features from images, SIFT allows machines to match those features to new images of the same object. This paves the way for exciting applications in image recognition, object detection, and more. In this article, we'll delve into the SIFT algorithm and explore how it empowers machines with human-like image recognition capabilities.

Learning Objectives

- A beginner-friendly introduction to the powerful SIFT (Scale Invariant Feature Transform) technique.
- Learn how to perform Feature Matching using the scale invariant feature transform algorithm.
- Try hands-on coding of the SIFT(scale invariant feature transform) algorithm in Python

6. Keypoint Localization
    - Local Maxima and Local Minima
    - Keypoint Selection
7. Orientation Assignment
    - Calculate Magnitude and Orientation
    - Creating a Histogram for Magnitude and Orientation
8. Keypoint Descriptor
9. Feature Matching

What is SIFT Algorithm?

The SIFT (Scale-Invariant Feature Transform) algorithm is a computer vision technique used for feature detection and description. It detects distinctive key points or features in an image that are robust to changes in scale, rotation, and affine transformations. SIFT(scale invariant feature transform) works by identifying key points based on their local intensity extrema and computing descriptors that capture

Take a look at the below collection of images and think of the common element between them:

The resplendent Eiffel Tower, of course! The keen-eyed among you will also have noticed that each image has a different background, is captured from different angles, and also has different objects in the foreground (in some cases).

Human vs. Machine Recognition

I'm sure all of this took you a fraction of a second to figure out. It doesn't matter if the image is rotated at a weird angle or zoomed in to show only half of the Tower. This is primarily because you have seen the images of the Eiffel Tower multiple times, and your memory easily recalls its features. We naturally understand that the scale or angle of the image may change, but the object remains the same.

But machines have an almighty struggle with the same idea. It's a challenge for

Applications of SIFT in Computer Vision

This is one of the most exciting aspects of working in computer vision!

> SIFT computer vision, or Scale Invariant Feature Transform, is a feature detection algorithm in Computer Vision.

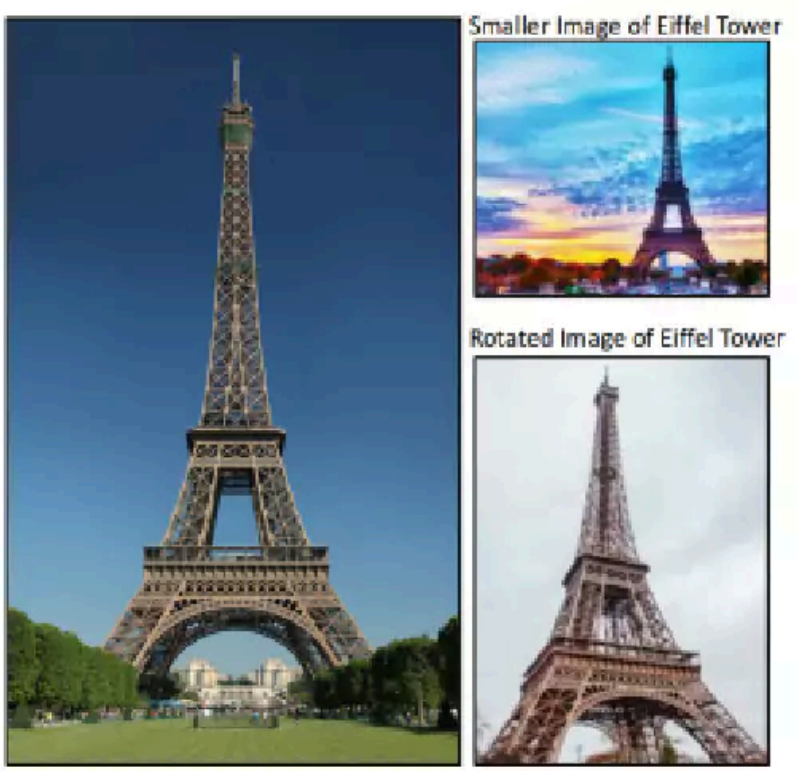
SIFT algorithm helps locate the local features in an image, commonly known as the '*keypoints*' of the image. These keypoints are scale & rotation invariants that can be used for various computer vision applications, like image matching, object detection, scene detection, etc.

Advantages of SIFT Keypoints

We can also use the keypoints generated using SIFT as features for the image during model training. The major advantage of SIFT features, over-edge features, or HOG features is that they are not affected by the size or orientation of the image.

Example Demonstration

For example, here is another image of the Eiffel Tower along with its smaller version. The keypoints of the object in the first image are matched with the keypoints found in the second image. The same goes for two images when the object in the other image is slightly rotated. Amazing, right?



The SIFT Process

Let’s understand how these keypoints are identified and what the techniques used to ensure the scale and rotation invariance are. Broadly speaking, the entire process can be divided into 4 parts:

- **Constructing a Scale Space:** To make sure that features are scale-independent
- **Keypoint Localisation:** Identifying the suitable features or keypoints
- **Orientation Assignment:** Ensure the keypoints are rotation invariant
- **Keypoint Descriptor:** Assign a unique fingerprint to each keypoint

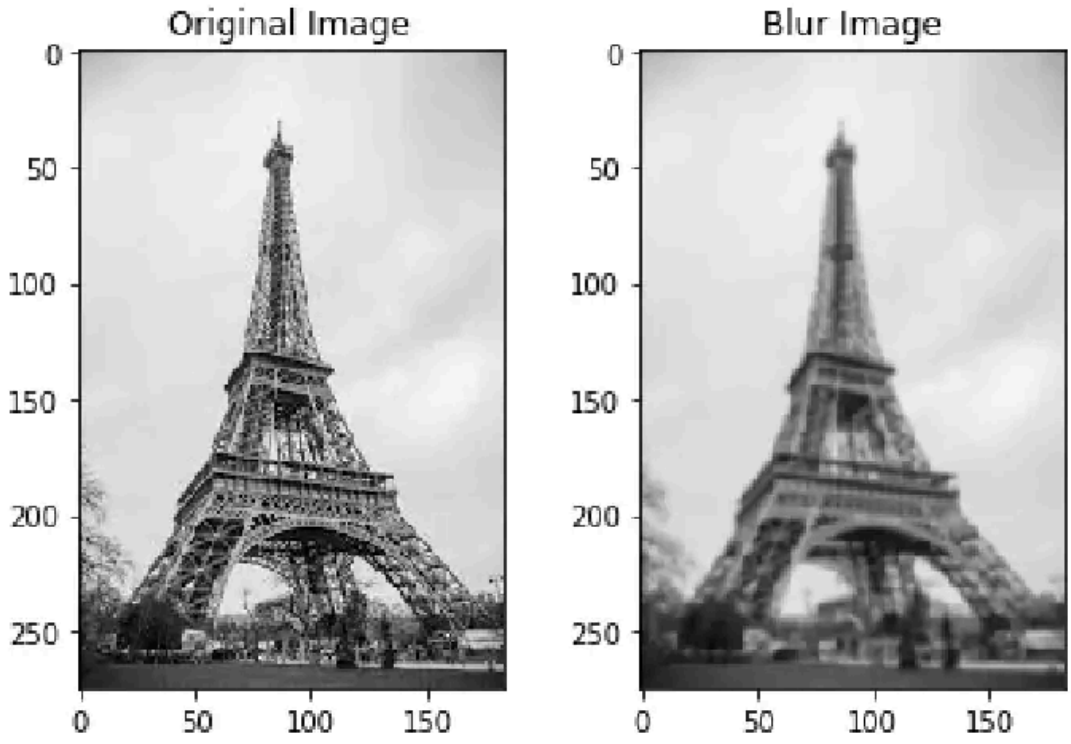
Finally, we can use these keypoints for feature matching!

dependent. These are critical concepts, so let's talk about them one by one.

Gaussian Blur

> We use the **Gaussian Blurring technique** to reduce the noise in an image.

For every pixel in an image, the Gaussian Blur calculates a value based on its neighboring pixels with a certain sigma value. Below is an example of an image before and after applying the Gaussian Blur. As you can see, the texture and minor details are removed from the image, and only the relevant information, like the shape and edges, remain:

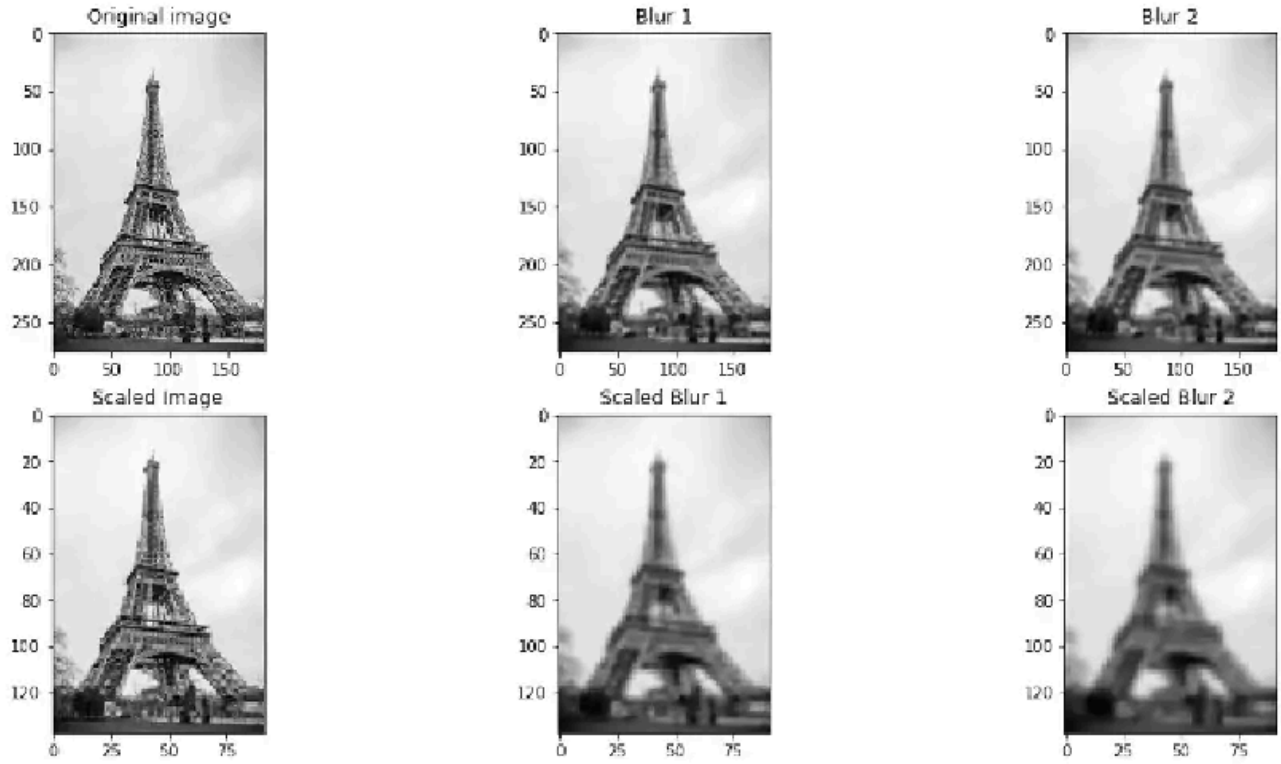


Gaussian Blur helped in image processing and successfully removed the noise from

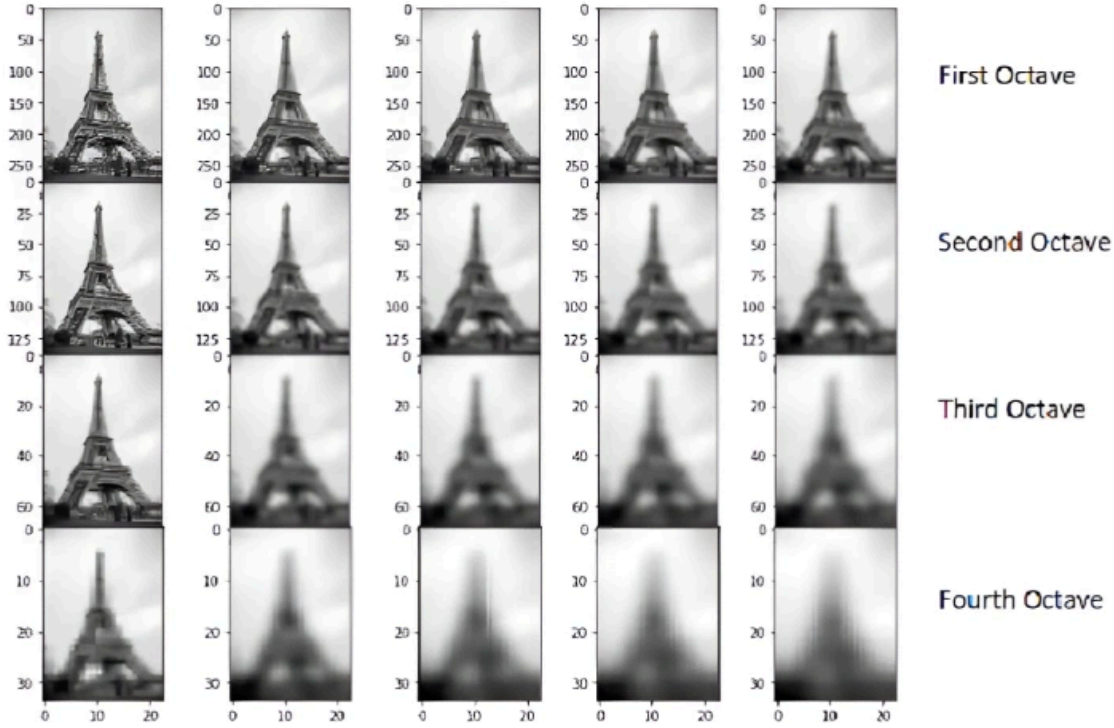
> Scale space is a collection of images having different scales, generated from a single image.

Hence, these blur images are created for multiple scales. To create a new set of images of different scales, we will take the original image and reduce the scale by half. For each new image, we will create blur versions as we saw above.

Here is an example to understand it in a better manner. We have the original image of size (275, 183) and a scaled image of dimension (138, 92). For both images, two blur images are created:



You might be thinking – how many times do we need to scale the image, and how many subsequent blur images need to be created for each scaled image? **The ideal number of octaves should be four**, and for each octave, the number of blur

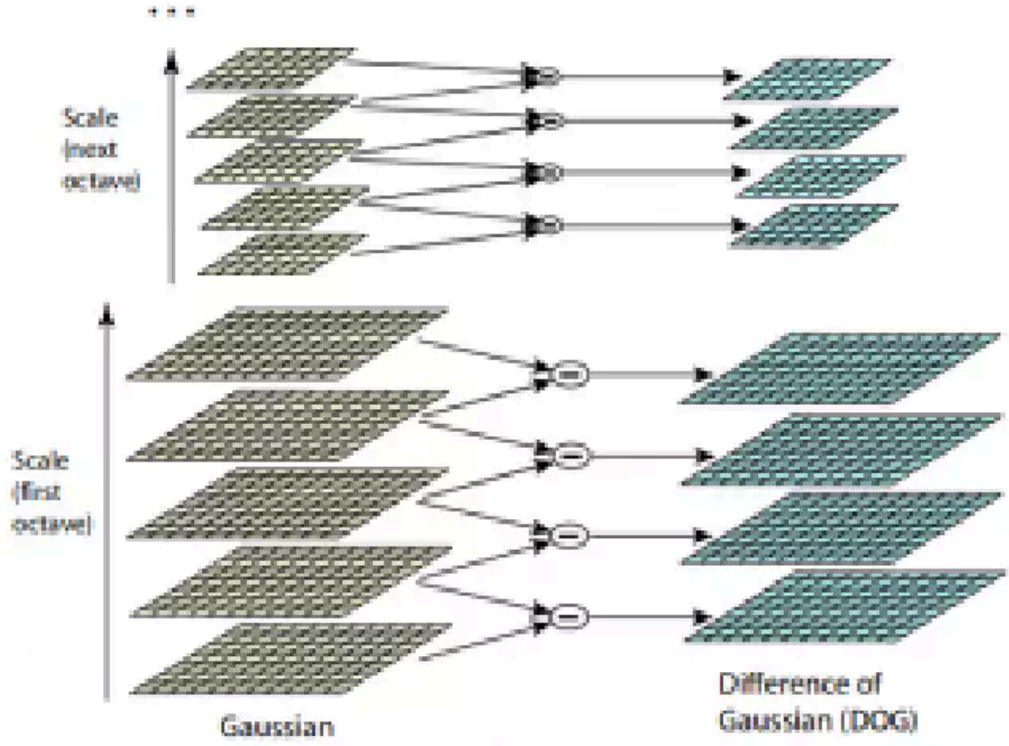


Difference of Gaussians

So far, we have created images of multiple scales (often represented by σ) and used Gaussian blur for each of them to reduce the noise in the image. Next, we will try to enhance the features using a technique called the Difference of Gaussians or DoG.

> Difference of Gaussian is a feature enhancement algorithm that involves the subtraction of one blurred version of an original image from another, less blurred version of the original.

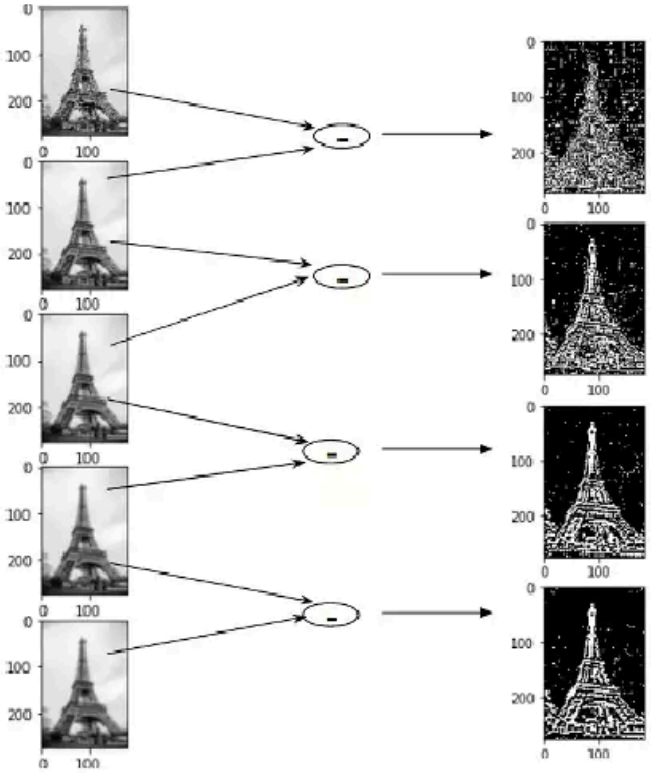
DoG creates another set of images, for each octave, by subtracting every image from the previous image in the same scale. Here is a visual explanation of how DoG is implemented:



Note: The image is taken from the original paper. The octaves are now represented in a vertical form for a clearer view.

Let us create the DoG for the images in scale space. Take a look at the below diagram. On the left, we have 5 images, all from the first octave (thus having the same scale). Each subsequent image is created by applying the Gaussian blur over the previous image.

On the right, we have four images generated by subtracting the consecutive Gaussians. The results are jaw-dropping!



We have enhanced features for each of these images. Note that here I am implementing it only for the first octave, but the same process happens for all the octaves.

Now that we have a new set of images, we are going to use this to find the important keypoints.

Keypoint Localization

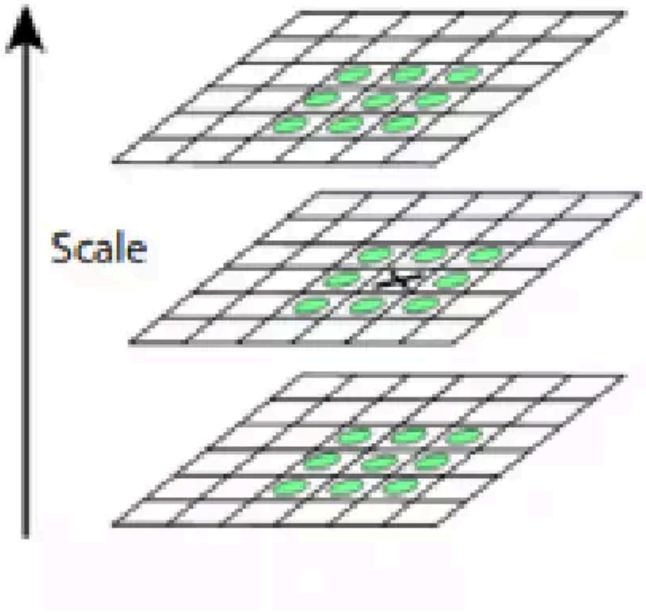
Once the images have been created, the next step is to find the important keypoints from the image that can be used for feature matching. **The idea is to find the local maxima and minima for the images.** This part is divided into two steps:

- Find the local maxima and minima
- Remove low contrast keypoints (keypoint selection)

> To locate the local maxima and minima, we go through every pixel in the image and compare it with its neighboring pixels.

When I say ‘neighboring’, this includes not only the surrounding pixels of that image (in which the pixel lies) but also the nine pixels for the previous and next image in the octave.

This means that every pixel value is compared with 26 other pixel values to find whether it is the local maxima/minima called extrema. For example, in the below diagram, we have three images from the first octave. The pixel marked *x* is compared with the neighboring pixels (in green) and is selected as a keypoint or interest point if it is the highest or lowest among the neighbors:



We now have potential keypoints that represent the images and are scale-invariant.

Kudos! So far, we have successfully generated scale-invariant keypoints. But some of these keypoints may not be robust to noise. This is why we need to perform a final check to make sure that we have the most accurate keypoints to represent the image features.

Hence, we will eliminate the keypoints that have low contrast or lie very close to the edge.

To deal with the low contrast keypoints, a second-order Taylor expansion is computed for each keypoint. If the resulting value is less than 0.03 (in magnitude), we reject the keypoint.

So what do we do about the remaining keypoints? Well, we perform a check to identify the poorly located keypoints. These are the keypoints that are close to the edge and have a high edge response but may not be robust to a small amount of noise. A second-order Hessian matrix is used to identify such keypoints. You can go through the math behind this here.

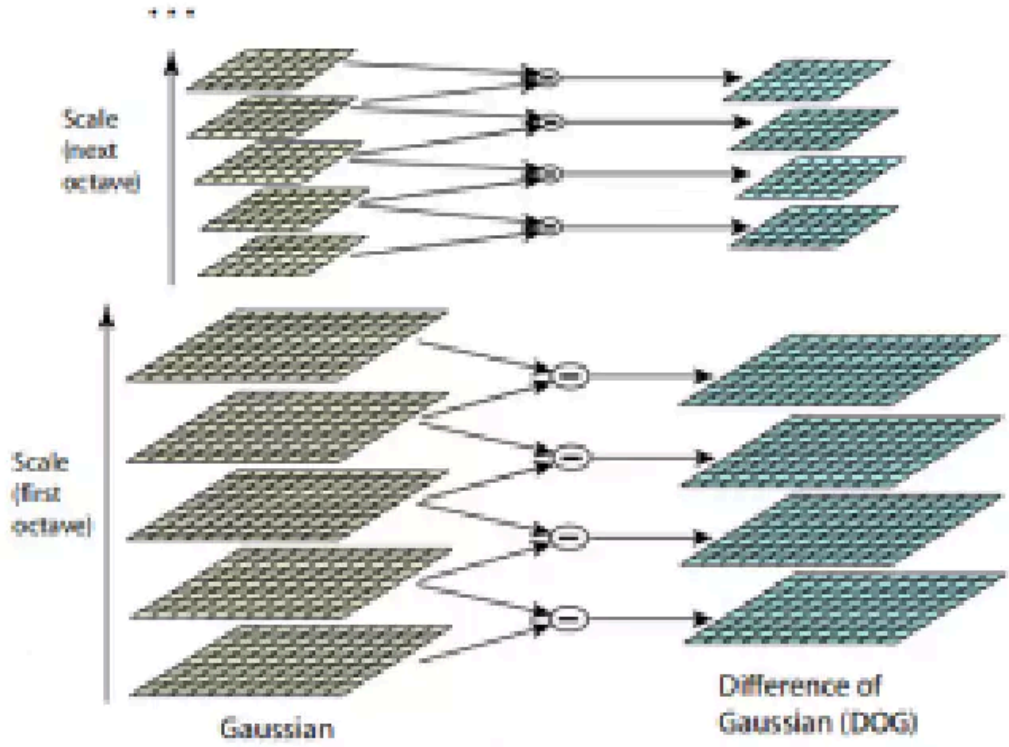
Now that we have performed both the contrast test and the edge test to reject the unstable keypoints, we will now assign an orientation value for each keypoint to make the rotation invariant.

Orientation Assignment

At this stage, we have a set of stable keypoints for the images. We will now assign an orientation to each of these keypoints so that they are invariant to rotation. We can again divide this step into two smaller steps:

Consider the sample image shown below:

35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75



Let’s say we want to find the magnitude and orientation for the pixel value in red. For

Once we have the gradients, we can find the magnitude and orientation using the following formulas:

Magnitude = $\sqrt{[(G_x)^2+(G_y)^2]}$ = 16.64

Φ = atan(Gy / Gx) = atan(1.55) = 57.17

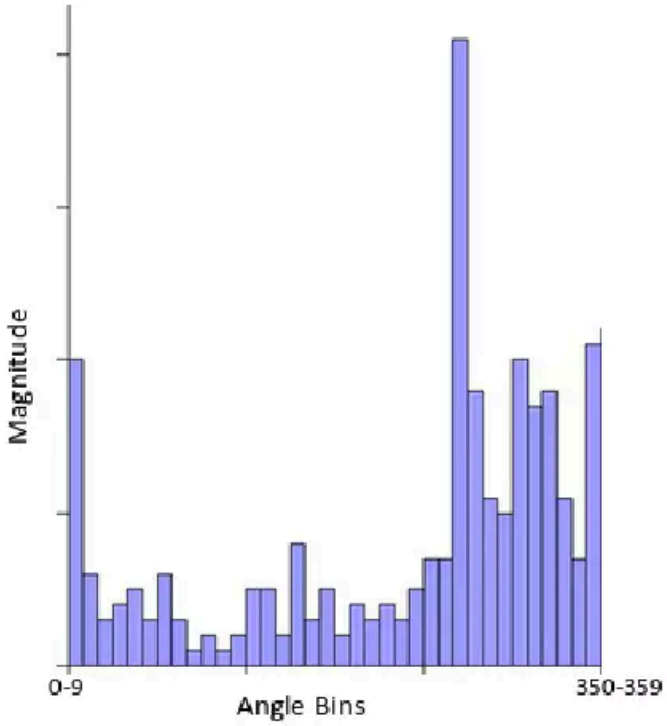
> The magnitude represents the intensity of the pixel and the orientation gives the direction for the same.

We can now create a histogram given that we have these magnitude and orientation values for the pixels.

Creating a Histogram for Magnitude and Orientation

On the x-axis, we will have bins for angle values, like 0-9, 10 – 19, 20-29, and up to 360. Since our angle value is 57, it will fall in the 6th bin. The 6th bin value will be in proportion to the magnitude of the pixel, i.e. 16.64. We will do this for all the pixels around the keypoint.

This is how we get the below histogram:



You can refer to this article for a much more detailed explanation for calculating the gradient, magnitude, orientation, and plotting histogram – A Valuable Introduction to the Histogram of Oriented Gradients.

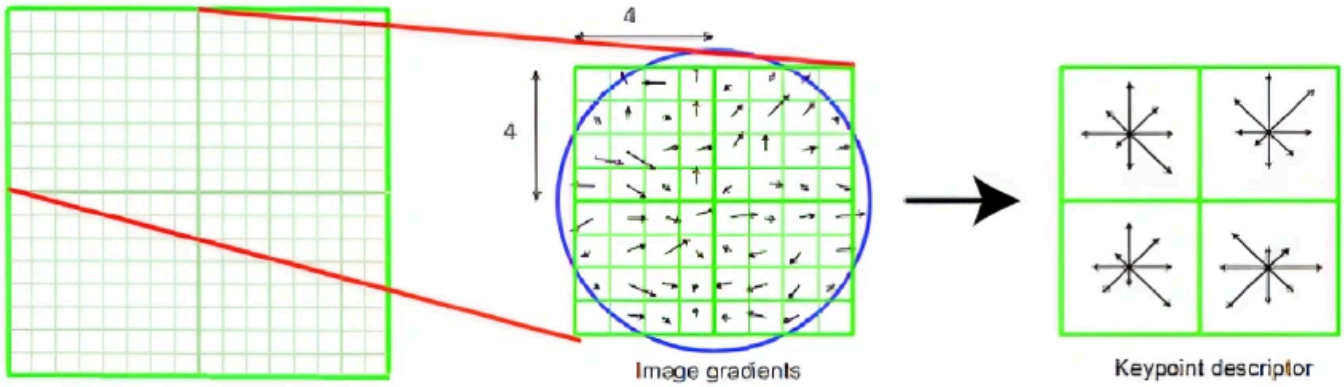
This histogram would peak at some point. **The bin at which we see the peak will be the orientation for the keypoint.** Additionally, if there is another significant peak (seen between 80 – 100%), then another keypoint is generated with the magnitude and scale the same as the keypoint used to generate the histogram. And the angle or orientation will be equal to the new bin that has the peak.

Effectively at this point, we can say that there can be a small increase in the number of keypoints.

Keypoint Descriptor

Additionally, since we use the surrounding pixels, the descriptors will be partially invariant to the illumination or brightness of the images.

We will first take a 16×16 neighborhood around the keypoint. This 16×16 block is further divided into 4×4 sub-blocks and for each of these sub-blocks, we generate the histogram using magnitude and orientation.



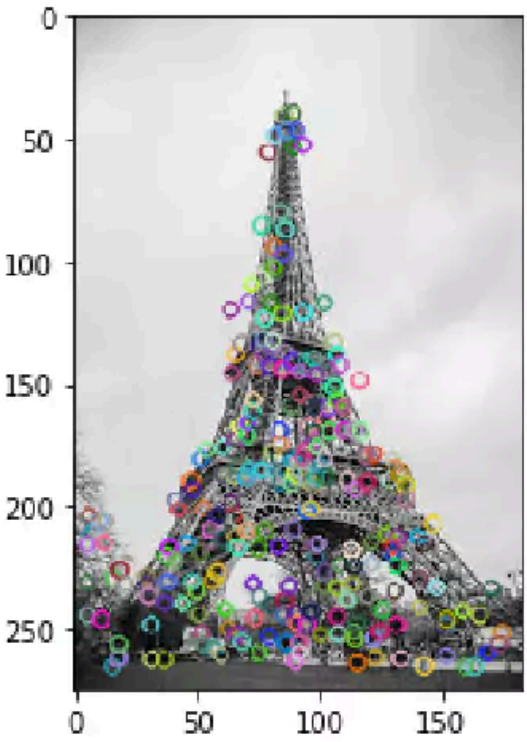
At this stage, the bin size is increased, and we take only 8 bins (not 36). Each of these arrows represents the 8 bins, and the length of the arrows defines the magnitude. So, we will have a total of 128 bin values for every keypoint.

Here is an example using pyplot in matplotlib:

Copy Code

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

```
img_1 = cv2.drawKeypoints(gray1,keypoints_1,img1)
plt.imshow(img_1)
```



Feature Matching

We will now use the SIFT computer vision features for feature matching. For this purpose, I have downloaded two images of the Eiffel Tower taken from different positions. You can try it with any two images that you want.

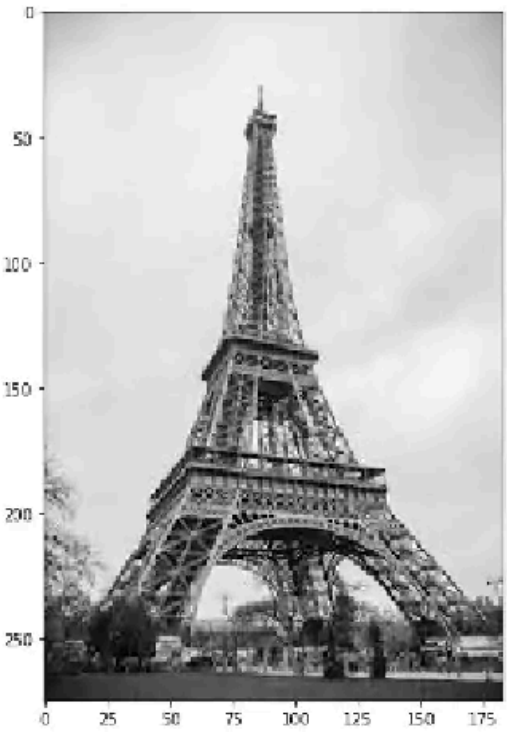
Here are the two images that I have used:

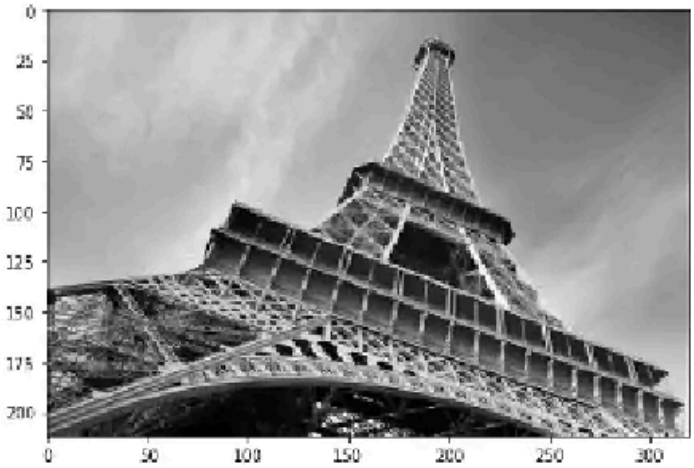
Copy Code

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# read images
```

```
ax[0].imshow(img1, cmap='gray')
ax[1].imshow(img2, cmap='gray')
```





Now, for both of these images, we are going to generate the SIFT features. First, we have to construct a SIFT object. We first create a SIFT computer vision object using sift_create and then use the function *detectAndCompute* to get the keypoints. It will return two values – the keypoints and the sift computer vision descriptors.

Let's determine the keypoints and print the total number of keypoints found in each image:

Copy Code

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# read images
img1 = cv2.imread('eiffel_2.jpeg')
```

```
sift = cv2.xfeatures2d.SIFT_create()

keypoints_1, descriptors_1 = sift.detectAndCompute(img1,None)
keypoints_2, descriptors_2 = sift.detectAndCompute(img2,None)

len(keypoints_1), len(keypoints_2)
```

Copy Code

```
283, 540
```

Next, let's try and match the features from image 1 with features from image 2. We will be using the function *match()* from the *BFmatcher* (brute force match) module. Also, we will draw lines between the features that match both images. This can be done using the *drawMatches* function in OpenCV python.

Copy Code

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# read images
img1 = cv2.imread('eiffel_2.jpeg')
img2 = cv2.imread('eiffel_1.jpg')

img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

#sift
sift = cv2.xfeatures2d.SIFT_create()

keypoints_1, descriptors_1 = sift.detectAndCompute(img1,None)
keypoints_2, descriptors_2 = sift.detectAndCompute(img2,None)

#feature matching
bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)

matches = bf.match(descriptors_1,descriptors_2)
matches = sorted(matches, key = lambda x:x.distance)
```

I have plotted only 50 matches here for clarity's sake. You can increase the number according to what you prefer. To find out how many keypoints are matched, we can print the length of the variable *matches*. In this case, the answer would be 190.

Conclusion

In this article, we discussed the SIFT feature-matching algorithm in detail. This site provides excellent visualization for each step of SIFT. You can add your own image, and it will create the keypoints for that image as well. Another popular feature-matching algorithm is SURF (Speeded Up Robust Feature), a faster version of SIFT. I encourage you to explore it as well.

And if you're new to the world of computer vision and image data, I recommend checking out the below course: Computer Vision using Deep Learning 2.0

Key Takeaways

- SIFT (Scale-Invariant Feature Transform) is a powerful technique for image matching that identifies and matches features invariant to scaling, rotation, and affine distortion.
- It sees widespread use in computer vision applications, including image matching, object recognition, and 3D reconstruction.
- The SIFT technique generates a scale space of images with different scales and then uses the Difference of Gaussian (DoG) method to identify keypoints.
- It also computes descriptors for each keypoint, which can be used for feature matching and object recognition.

A. SIFT and SURF are both feature detection algorithms in computer vision. The main difference is that SURF is faster and more computationally efficient than SIFT, but SIFT is generally more robust to various image transformations.

Q2. Can you draw only the keypoints or the keypoints with descriptor vectors as well in SIFT?

A. In SIFT computer vision, you can draw both the keypoints and the descriptor vectors. Using the `drawKeypoints` function in OpenCV, set the `flags` parameter to `cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS` to visualize the keypoints with their size, orientation, and descriptor vectors as arrows.

Q3. Is grayscale a necessary step in SIFT?

A. Yes, you must convert images to grayscale in SIFT computer vision because the algorithm works with single-channel grayscale images. Without this step, the algorithm cannot accurately detect keypoints and compute descriptors.

Q4.What is SIFT in CV?

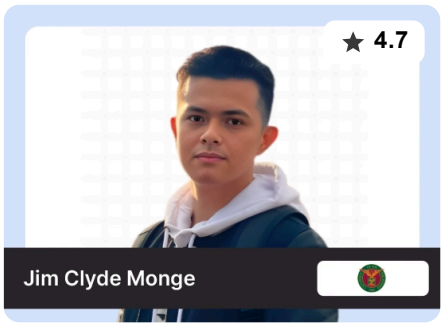
SIFT is a computer vision algorithm used for detecting and describing unique features in images. These features are invariant to scale, rotation, and illumination. They are used for tasks like object recognition, image matching, and 3D reconstruction.

Aishwarya Singh

An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.

Beginner | Computer Vision | Image | Python | Python | Technique | Unstructured Data

Free Courses

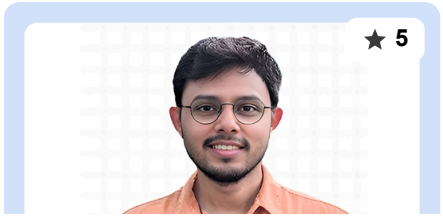


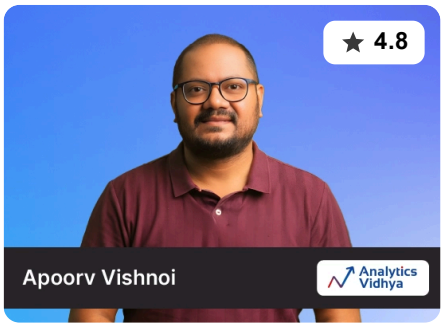
How to Build an Image Generator Web App with Zero Coding

Learn to build an image generator web app with zero coding skills.

Mastering Multimodal RAG & Embeddings with Amazon Nova & Bedrock

Master multimodal RAG and embeddings using Amazon Nova and Bedrock.





Building Your First Computer Vision Model

Build your first computer vision model with Pytorch.

Responses From Readers

What are your thoughts?...

Submit reply

Matang Panchal

You have made this article very simple to understand, Great Work. One Question , How can we decide that images are matching or not, Because as shown in the last image few keypoints are matching and few are not matching, so what would be the deciding factor?

Thanks for a very interesting and informative article. Is it possible to publish links to the image files to allow readers to easily try out your code?

1 Show 1 reply

sasi reddy

Frequently Asked Questions

Q1. What is the difference between SIFT and SURF?

A. SIFT and SURF are both feature detection algorithms in computer vision. The main difference is that SURF is faster and more computationally efficient than SIFT, but SIFT is generally more robust to various image transformations.

Q2. Can you draw only the keypoints or the keypoints with descriptor vectors as well in SIFT?

Q3. Is grayscale a necessary step in SIFT?

Q4.What is SIFT in CV?

Flagship Programs

GenAI Pinnacle Program | GenAI Pinnacle Plus Program | AI/ML BlackBelt Program | Agentic AI Pioneer Program

Free Courses

Generative AI | DeepSeek | OpenAI Agent SDK | LLM Applications using Prompt Engineering | DeepSeek from Scratch | Stability.AI | SSM & MAMBA | RAG Systems using LlamaIndex | Building LLMs for Code | Python | Microsoft Excel | Machine Learning | Deep Learning | Mastering Multimodal RAG | Introduction to Transformer Model | Bagging & Boosting | Loan Prediction | Time Series Forecasting | Tableau | Business Analytics | Vibe Coding in Windsurf | Model Deployment using FastAPI | Building Data Analyst AI Agent | Getting started with OpenAI o3-mini | Introduction to Transformers and Attention Mechanisms

Popular Categories

AI Agents | Generative AI | Prompt Engineering | Generative AI Application | News | Technical Guides | AI Tools | Interview Preparation | Research Papers | Success Stories | Quiz | Use Cases | Listicles

Generative AI Tools and Techniques

GANs | VAEs | Transformers | StyleGAN | Pix2Pix | Autoencoders | GPT | BERT | Word2Vec | LSTM | Attention Mechanisms | Diffusion Models | LLMs | SLMs | Encoder Decoder Models | Prompt Engineering | LangChain | LlamaIndex | RAG | Fine-tuning | LangChain AI Agent | Multimodal Models | RNNs | DCGAN | ProGAN | Text-to-Image Models | DDPM | Document Question Answering | Imagen | T5 (Text-to-Text Transfer Transformer) | Seq2seq Models | WaveNet | Attention Is All You Need (Transformer Architecture) | WindSurf | Cursor

Popular GenAI Models

Llama 4 | Llama 3.1 | GPT 4.5 | GPT 4.1 | GPT 4o | o3-mini | Sora | DeepSeek R1 | DeepSeek V3 | Janus Pro | Veo 2 | Gemini 2.5 Pro | Gemini 2.0 | Gemma 3 | Claude Sonnet 3.7 | Claude

n8n | LangChain | Agent SDK | A2A by Google | SmolAgents | LangGraph | CrewAI | Agno | LangFlow | AutoGen | LlamaIndex | Swarm | AutoGPT

Data Science Tools and Techniques

Python | R | SQL | Jupyter Notebooks | TensorFlow | Scikit-learn | PyTorch | Tableau | Apache Spark | Matplotlib | Seaborn | Pandas | Hadoop | Docker | Git | Keras | Apache Kafka | AWS | NLP | Random Forest | Computer Vision | Data Visualization | Data Exploration | Big Data | Common Machine Learning Algorithms | Machine Learning | Google Data Science Agent